Kilo TM Correctness: ABA Tolerance and Validation-Commit Indivisibility

Wilson W. L. Fung Inderpreet Singh Tor M. Aamodt
wwlfung@ece.ubc.ca isingh@ece.ubc.ca aamodt@ece.ubc.ca

Department of Computer and Electrical Engineering
University of British Columbia

May 24, 2012

1 Summary

Kilo TM is a hardware transactional memory (TM) system proposed for GPU architectures [1]. In Kilo TM, each transaction detects the existence of conflicts with other transactions via *value-based conflict detection* [2, 3]. With value-based conflict detection, each transaction buffers its writes to memory in a write-log and saves the values of its reads from memory in a read-log during execution. Upon its completion, the transaction compares the saved values of its read-set with the latest values in memory before it commits. We refer this comparison as *validation*. Any difference between the saved value and the latest value in memory indicates the existence of a conflict. Kilo TM uses value-based conflict detection because it avoids direct communication between transactions, and it does not require any essential on-chip storage (Kilo TM uses on-chip storage to improve commit parallelism). More details regarding the design of Kilo TM are available in our paper for the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2011) [1].

A general concern for the correctness of value-based conflict detection is the possibility of subtle bugs due to the ABA problem. We show that value-based conflict detection can tolerate the ABA problem, and like NOrec [3], we can create a logical order for Kilo TM in which validation and commit of each transaction are indivisible.

1.1 Tolerance to the ABA Problem

Examples of ABA problems have been found for published non-blocking algorithms. These generally result from an implicit assumption that atomicity of a high-level operation on a concurrent data structure can be inferred as long as the value of a guard variable is the

same after a sequence of low-level instructions (that implements the operation). This fallacy results in subtle bugs [4].

We argue a TM system with value-based conflict detection is not vulnerable to the ABA problem as long as it ensures that each transaction's read-set has observed a consistent view of memory (i.e. conflict detection is not comparing values with a partially committed transaction).

Consider a TM system with address-based conflict detection (full knowledge of which locations have been modified by other transactions). If any location read by a committing transaction has been changed and restored to its original value since the transaction originally read it, aborting the transaction and rerunning it instantaneously with the updated memory would yield the same result (same addresses and values in its write-set). This situation summarizes the behavior of a transaction in Kilo TM that has passed value-based conflict detection where other conflicting transactions changed and then restored values during the transaction execution. Committing the transaction directly without rerunning it effectively serializes it behind the last committed transaction. This requires the transaction's read-set to observe a consistent view of memory and commit before other transactions update locations in the transaction's write-set, which is achieved as shown below.

1.2 Logical Validation-Commit Indivisibility

In Kilo TM, each transaction T_X is given a unique commit ID prior to validation and commit, and this ID defines the *commit order* of T_X. T_X will obtain a new commit ID for each execution attempt (i.e. a new ID is assigned every time T_X was aborted). Given two transactions T_X and T_Y with commit ID X and Y, where $X < Y$, Kilo TM's implementation guarantees the following partial ordering:

- Validation of each word w by T_Y always happens after any write to w by T_X.
- Any write to w by T_Y always happens after any write to w by T_X.
- Validation of w by T_X always happens before any write to w by T_Y.

These guarantees order validations and writes at each memory location (word) in ascending commit order.

With this per-word order, it can be shown that transactions committed by Kilo TM satisfy *conflict serializability* [5]: All conflict relations produced from the accesses at each word will obey the commit order. Hence, a conflict graph created from these conflict relations is always acyclic.

Conflict serializability implies a logical timeline in which validation and commit of each transaction are indivisible (performed without being interleaved by other transactions). The logical timeline also implies that validation of each transaction always observes a consistent view of memory.

2 Outline of Proof

The remainder of this document contains a proof of Kilo TM correctness. The proof begins with defining a memory value-location framework that is used to represent the values observed and produced by each transaction. This framework is then used to model a general

ABA problem scenario specific to TM systems that employ value-based conflict detection. The proof for Theorem 1 uses this framework to show that such TM systems can tolerate the ABA problem, under the assumptions that validation and commit of each transaction are indivisible, and validation is done on a consistent view of memory. To show that Kilo TM satisfies these assumptions, Claim 1 to 6 are used to prove Lemma 1 and Lemma 2, which together show that accesses from transactions to each word are ordered by ascending commit IDs. From this per-word ordering, Lemma 3 shows that validation performed by each transaction accesses a consistent view of memory. Then, Lemma 4 shows that accesses from transactions satisfy *conflict serializability* [5], which means there exists a logical timeline in which validation and commit of each transaction is indivisible. Finally, Theorem 2 combines Lemma 3 and Lemma 4 to show that Kilo TM satisfies the assumptions for Theorem 1. Therefore, Kilo TM can tolerate the ABA problem.

3 Memory Value-Location Framework for the ABA Problem

Transactional memory (TM) provides the programmer with the abstraction that transactions are executed in some serialization order. In this serialization order, transactions are executed serially one after another. Each transaction T advances the memory state from the original state observed by T, M^O, to a new state M^N. We denote this transition from M^O to M^N with $M^O \rightarrow M^N$.

Each memory state is defined by the value at every location in the entire memory space M (i.e. a mapping from each address in M to its value). A memory state M^X is equivalent to another memory state M^Y if the value at each location in M^X is equal to the value at the corresponding location in M^Y. For the rest of this discussion, we use subscripts to denote subsets of locations within a memory space and superscripts to denote different memory states (i.e. values at a set of memory locations). For example, M_R is a subset of memory locations (not values) of the memory space M, M^L is the values in every location in the entire memory space M, and M_R^L is the set of address-value pairs for the memory locations (addresses) in M_R with values from the memory state M^L. We call M_R^L a partial memory state.

For each transaction T_X, the entire memory space M is divided into the following subsets:

$M_{R,X} =$ The memory locations read by T_X as it executes (Read-Set).

$M_{W,X} =$ The memory locations written by T_X when it commits (Write-Set).

$M_{A,X} = M_{R,X} \cup M_{W,X} =$ The memory locations that are accessed (either read or written) by T_X.

$M_{I,X} = M - M_{A,X} =$ Memory locations that are ignored (neither read nor written) by T_X.

$M_{RW,X} = M_{R,X} \cap M_{W,X} =$ Memory locations in T_X's Read-Set that are also part of its Write-Set.

$M_{RO,X} = M_{R,X} - M_{RW,X} =$ Memory locations that are only read by T_X.

$M_{WO,X} = M_{W,X} - M_{RW,X} =$ Memory locations that are only written by T_X.

Notice $M_{I,X}$, $M_{RW,X}$, $M_{RO,X}$ and $M_{WO,X}$ are all disjoint sets, and $M = M_{I,X} \cup M_{RW,X} \cup M_{RO,X} \cup M_{WO,X}$.

During execution, each transaction T_X observes the partial memory state $M^O_{R,X} = M^O_{RO,X} \cup M^O_{RW,X}$, and writes to addresses in $M_{W,X}$, producing the new partial memory state $M^N_{W,X} = M^N_{WO,X} \cup M^N_{RW,X}$. In the mean time, other transactions may have committed, advancing the latest global memory states to $M^L = M^L_{R,X} \cup M^L_{W,X} \cup M^L_{I,X}$. With value-based conflict detection, the transaction checks to see if $M^L_{R,X} = M^O_{R,X}$ before it commits. If the two partial memory states are indeed equivalent, T_X commits by advancing the partial memory state in its write-set from $M^L_{W,X}$ to $M^N_{W,X}$. This appends the serialization order with a new transition $(M^L \rightarrow M^N)$.

3.1 ABA Problem

The ABA problem manifests in non-blocking algorithms, where multiple threads may operate on a data structure simultaneously, and the atomicity of each operation is presumed to be guaranteed via success of one or more atomicCAS operations. Many non-blocking algorithms rely on the following assumption: If the value of a guarding variable has not been modified since it was last read, then no other threads have modified the data structure, and thus this thread has performed the current operation in isolation. This assumption ignores the possibility that several other operations may have occurred in between, first modifying the guard variable to other values, then restoring the original value before the current thread uses atomicCAS to check the variable's value. The fallacy in this assumption is how the ABA problem manifests in various non-blocking algorithms, resulting in subtle bugs that are hard to detect [4, 6].

3.2 Potential ABA Problem in Transactional Memory

In the context of value-based conflict detection employed in Kilo TM, we consider the potential for ABA problems in the following form. A set of transactions $T_{ABA} = \{T_1, \ldots T_L\}$ have committed in between time t_1, when transaction T_X first started to read its read-set $M_{R,X}$ (observing state $M^O_{R,X}$ from M^O, the memory state before any transaction in T_{ABA} commits), and the time t_2, when T_X is validating its read-set against the latest global memory state. We assume that transactions are weakly isolated [7] and ignore non-transactional writes in this discussion. Transactions in T_{ABA} advance the global memory state from M^O through a series of memory states and eventually to M^L. M^L is not necessarily equivalent to M^O, but the part that belongs to T_X's read-set is equivalent: $M^O_{R,X} = M^L_{R,X}$. Value-based conflict detection performed by T_X will observe that its read-set has not been changed, and T_X "assumes" no conflicting transaction has committed between t_1 and t_2 (i.e. the values in $M_{R,X}$ appear to have never been modified in this window). Subsequently, T_X commits by advancing $M^L_{W,X}$ to $M^N_{W,X}$, whereas the intended transition (one that would have occurred if T_X has executed in isolation without the presence of transactions in T_{ABA}) is from $M^O_{W,X}$ to $M^N_{W,X}$. This intended transition $(M^O \rightarrow M^N)$ violates the existing serialization order because the latest memory state is M^L. A TM system with address-based conflict detection will regard this as a conflict. In such a system, T_X will be restarted to resolve this conflict. However, we will show that this restart is not needed.

3.3 Tolerance to the ABA Problem

The following proof shows that committing T_X directly in the situation described in Section 3.2 will result in the same serialization order in which T_X detects the conflict and reruns itself starting at time t_2. In other words, TM systems employing value-based conflict detection can tolerate the ABA problem by yielding the same memory state transition as TM systems employing address-based conflict detection.

Theorem 1. *Directly committing T_X in the situation described in Section 3.2 will result in the same serialization order in which T_X detects the conflict and reruns itself instantly starting at time t_2.*

Proof. Assume that the TM system employs a separate mechanism other than value-based conflict detection that is not prone to the ABA problem. T_X, upon detecting the conflict at time t_2, aborts itself and restarts immediately. Let T_X^1 be this new instance of T_X. T_X^1 will observe its read-set from M^L. Since $M_{R,X}^L = M_{R,X}^O$, T_X^1 will produce the identical write-set partial memory state $M_{W,X}^N$ as T_X. If T_X^1 finishes executing instantaneously with no other transactions committing in between, its commit will advance the partial memory state in $M_{W,X}$ from $M_{W,X}^L$ to $M_{W,X}^N$ and T_X^1 will transition the global memory state from M^L to $M^N = (M_{I,X}^N \cup M_{RO,X}^N \cup M_{W,X}^N)$. Values in $(M_{I,X} \cup M_{RO,X})$ remain unchanged between M^L and M^N (i.e. $M_{I,X}^N = M_{I,X}^L$ and $M_{RO,X}^N = M_{RO,X}^L$). Committing T_X at time t_2 would have produced the same transition ($M^L \rightarrow M^N$): $M_{W,X}^L$ is advanced to $M_{W,X}^N$, while values in $(M_{I,X} \cup M_{RO,X})$ remain unchanged. Therefore, as committing either T_X or T_X^1 results in the same transition, the programmer cannot discern between the two instances of execution. □

The above proof made two assumptions:

Assumption 1. *T_X commits immediately after value-based conflict detection, such that no other transactions can commit in between to advance the memory state away from M^L.*

Assumption 2. *The value-based conflict detection performed by T_X is comparing the original read-set state $M_{R,X}^O$ against a consistent view of the global memory state $M_{R,X}^L$. Here consistent view means that during conflict detection, $M_{R,X}^L$ is not advanced to another memory state by the commit of another transaction (i.e. $M_{R,X}^L$ is the part of the memory state that exists in between the commits of two transactions).*

We proceed to demonstrate that Kilo TM, despite its distributed design, satisfies both assumptions.

3.4 Inconsistent Read-Set

While T_X can possibly have observed an inconsistent view of memory (e.g. partially committed states from transactions in T_{ABA}) during its execution, Theorem 1 holds as long as T_X's observed read-set equals to $M_{R,X}^L$. T_X may also observe inconsistent values from a single memory location. In Kilo TM, each transactional load appends the value read from global memory into a linear read-log (if it is not accessing the transaction's write-set) [1]. The inconsistent values observed from a single memory location by T_X will create multiple read-log entries that contain different values for a single location. During value-based conflict

detection, only one of the values will match with the one in $M^L_{R,X}$. The mismatched entry will cause T_X's validation to fail (subsequently aborting T_X). T_X may enter an infinite loop due to the inconsistent view of memory. To ensure that T_X is eventually aborted, Kilo TM employs a watchdog timer to trigger a validation for T_X [1].

4 Transaction Components in Kilo TM

In Kilo TM, each transaction, T_X, is comprised of the following sequence of operations:

$$T_X = R(r_1) \dots R(r_m)\ Rv(r_1) \dots Rv(r_m)\ W(w_1) \dots W(w_n)$$

- $R(r_1)$ is a read operation from word r_1, and $M_{R,X} = \{r_1 \dots r_m\}$ is the read-set of T_X.
- $Rv(r_1)$ is a validation operation on word r_1, ensuring that the value obtained by R(r1) equals the value in global memory. (This is the operation that performs value-based conflict detection.)
- $W(w_1)$ is a write operation to word w_1, and $M_{W,X} = \{w_1 \dots w_n\}$ is the write-set of T_X. The write operation is performed only while T_X commits, and it updates the value of w_1 in global memory.

When there are multiple transactions involved, the notation $Rv(T_X, w)$ denotes the validation operation on word w for transaction T_X; whereas notation $Rv(T_X)$ denotes all of the validation operations required by T_X.

T_X is executed on a single SIMT core [1]. The core interacts with a set of commit units (one in each memory partition) to validate and commit T_X. The following messages are sent between the commit units and the core that executes T_X:

- $V_k(T_X)$ = (pass/fail) is the validation outcome for T_X at commit unit k. $V_k(T_X)$ = pass if validation operations performed on each word w contained in commit unit k for T_X, where $w \in M_{R,X}$, all succeed; otherwise, $V_k(T_X)$ = fail. This message is sent from each commit unit k to the core running T_X.
- $F(T_X)$ = (pass/fail) is the final outcome for T_X. $F(T_X)$ = pass if all validation outcomes received by the core $V_k(T_X)$ = pass; otherwise, $F(T_X)$ = fail. This message is sent from the core to each commit unit. Write operations $W(T_X)$ are only performed if $F(T_X)$ = pass.

4.1 Commit ID and Commit Order

Prior to validation and commit, a transaction T is given a unique commit ID. This ID defines the *commit order* of T. A transaction with lower commit ID has an earlier commit order than those with a higher commit ID. Namely, given transactions T_X and T_Y with commit ID X and Y respectively, $X < Y \iff T_X <_t T_Y$. Here $<_t$ denotes the commit order. Each transaction will obtain a new commit ID for each execution attempt (i.e. a new ID is assigned every time the transaction was aborted).

The commit order limits how a transaction may appear in the serialization order. Let M^L be the latest memory state, and T_X and T_Y be two transactions, with $T_X <_t T_Y$, that are ready to commit. Only one of the following can happen:

- Both T_X and T_Y commit, resulting in transitions $M^L \rightarrow M^{X1} \rightarrow M^{Y2}$, where M^{X1} is the memory state from M^L after T_X has committed and M^{Y2} is the memory state from M^{X1} after T_Y has committed.
- Only T_X commits, resulting in transition $M^L \rightarrow M^{X1}$, where M^{X1} is the memory state from M^L after T_X has committed.
- Only T_Y commits, resulting in transition $M^L \rightarrow M^{Y1}$, where M^{Y1} is the memory state from M^L after T_Y has committed.
- Neither T_X nor T_Y commits, resulting in no transition.

Notice that transitions $M^L \rightarrow M^{Y1} \rightarrow M^{X2}$ (M^{X2} is the memory state from M^{Y1} after T_X has committed) is not allowed. This restriction is enforced by Claim 2 below. This allows Kilo TM to handle transactions with write-after-write conflicts by ordering their commits without synchronizing among different commit units (instead of aborting one of them). RingSTM [8] also uses this policy to handle write-after-write conflicts.

5 Partial Orderings Provided by Kilo TM

Kilo TM's implementation provides a set of partial orderings that we present in the following claims. The following discussions on the validity of these claims assume that the reader is familiar with the implementation of Kilo TM. An in-depth description of Kilo TM's implementation can be found in Section 4.5 of the MICRO 2011 paper [1]. These claims will be used to show that Kilo TM satisfies both Assumption 1 and Assumption 2 required for ABA problem tolerance (Theorem 1). We denote these partial orderings with $<_P$. Given two events/operations A and B, $A <_P B$ means A *happens before* B in real time.

Claim 1. *At each commit unit, given transactions T_X and T_Y, where $T_X <_t T_Y$, a write to a memory location w performed by a transaction T_X always happens before validation of the same location w performed by a transaction T_Y . In our notation, $W(T_X, w) <_P Rv(T_Y, w)$.*

Proof. In Kilo TM's implementation, transactions always perform hazard detection in commit order. Each commit unit can speculatively validate each memory location in the read-set of T_Y ($M_{R,Y}$) as the corresponding read-log entry arrives at the unit. Later in hazard detection, if the unit detects (via address-based conflict detection) that T_X is writing to any part of $M_{R,Y}$, the unit will revalidate the entire read-set of T_Y after T_X has finished committing. □

Claim 2. *Let T_X and T_Y be transactions with $T_X <_t T_Y$. If a memory location w is in the write-sets of both T_X and T_Y , writes to w by transaction T_X always happen before writes to w by transaction T_Y. Namely, $W(T_X, w) <_P W(T_Y, w)$.*

Proof. This is enforced at the commit stage in each commit unit. This ordering is guaranteed by issuing the write operations of each passed transaction in ascending commit order. The GPU memory subsystem in our architecture can reorder accesses to different locations to optimize for bandwidth, but it maintains the ordering of accesses to the same location. □

Claim 3. *At each commit unit, the write operations for a transaction T_X are only commenced after the commit unit has received the final outcome $F(T_X)$ of T_X from the core. Namely, $F(T_X) <_P W(T_X)$.*

Proof. This is enforced at the finalizing outcome stage in each commit unit. □

Claim 4. *The transaction will not send out the final outcome $F(T_X)$ to commit units until it has received validation outcomes $V_k(T_X)$ from all commit units for the transaction. For each commit unit k that contains any location in the read-set of T_X, $V_k(T_X) <_P F(T_X)$.*

Proof. This is enforced by the Kilo TM implementation at each SIMT core. □

Claim 5. *At each commit unit k that contains any location in the read-set of T_X, the validation outcome $V_k(T_X)$ is sent after all validation operations are done. Namely, $Rv(T_X) <_P V_k(T_X)$.*

Proof. This is enforced at the finalizing outcome stage in each commit unit. □

Claim 6. *At each commit unit, the write operations for a transaction T_X are only commenced after the commit unit has received the final outcomes $F(T_Y)$ from all transactions T_Y with earlier commit order if the commit unit contains any location in either read-set or write-set of T_Y ($M_{A,Y}$). I.e. For all transaction T_Y with ($T_Y <_t T_X$) and the commit unit that contains any location in $M_{A,Y}$, $F(T_Y) <_P W(T_X)$.*

Proof. This is enforced at the finalizing outcome stage in each commit unit. At each commit unit, the commit unit entry that corresponds to a transaction T_Y waits for the final outcome $F(T_Y)$ before proceeding to the commit stage. T_Y stalling at the finalizing outcome stage will forbid any transaction with a younger commit ID (e.g. T_X) to proceed to the next stage, even after $F(T_X)$ has been received by the unit. Since write operations are only issued in commit stage, this stalling behavior enforces Claim 6. □

6 Per-Word Access Ordering

Lemma 1 and 2 illustrate how Kilo TM orders accesses (validations and writes) to each memory location (word) in ascending commit order. This ordering is used in Lemma 3 to prove that all validation operations for a given transaction T_X are comparing against a consistent view of $M^L_{R,X}$ (Assumption 2).

Lemma 1. *Let T_Y be a transaction. For each memory location w in $M_{RW,Y}$, validation operation(s) to w by T_Y always happen before write operation(s) to w by T_Y.*

Proof. Let t_C be the time when T_Y starts sending the final outcome $F(T_Y)$ to each commit unit from the core. By Claim 5, all the validation operations of w by T_Y ($Rv(T_Y, w)$) at each commit unit k have to happen before the unit replies with the validation outcome $V_k(T_Y)$ back to the core. By Claim 4, all of these outcomes have to arrive at the core before T_Y sends out $F(T_Y)$, i.e. before t_C. The same commit unit k will receive $F(T_Y)$ at a time $t_K > t_C$. By Claim 3, any write operation to w in $M_{RW,Y}$ by T_Y ($W(T_Y, w)$) has to happen after t_K.

Putting it all together, all validation operations to w in $M_{RW,Y}$ by T_Y have to occur before t_C, which is before all write operations to w by T_Y. I.e. $Rv(T_Y, w) <_P t_C <_P t_K <_P W(T_Y, w) \Rightarrow Rv(T_Y, w) <_P W(T_Y, w)$. □

Lemma 2. *Let T_X and T_Y be transactions with $T_X <_t T_Y$. For each memory location w, operations (validation/write) to w by T_X always happen before operations to w by T_Y, except when both operations are validation.*

Proof. This can be broken down into three separate orderings:

O1. $W(T_X, w) <_P Rv(T_Y, w)$
O2. $W(T_X, w) <_P W(T_Y, w)$
O3. $Rv(T_X, w) <_P W(T_Y, w)$

The first two orderings follow directly from Claim 1 and Claim 2 respectively. The final ordering follows from Claim 4-6: At each commit unit k, the validation outcome of T_X, $V_k(T_X)$, is only sent after all its validation operations are done (Claim 5), and the final outcome $F(T_X)$ will only arrive after all validation outcomes have been received by the core (Claim 4). By Claim 6, the write operations of T_Y will not be issued until the commit unit has received $F(T_X)$. Putting it all together, let commit unit k be the unit containing location w, $Rv(T_X, w) <_P V_k(T_X) <_P F(T_X) <_P W(T_Y, w) \Rightarrow Rv(T_X, w) <_P W(T_Y, w)$. □

7 Validation Against a Consistent View of Memory

Lemma 3. *The value-based conflict detections (all validation operations) performed by T_Y compare the original read-set state $M^O_{R,Y}$ against a consistent view of a global memory state $M^L_{R,Y}$.*

Proof. Claim 2 (O2 in Lemma 2) specifies that each location in memory is written in ascending commit order. By O1 and O3 in Lemma 2, every validation by T_Y to a location w in $M_{R,Y}$ is performed after all transactions with earlier commit orders have written to w and before transactions with later commit orders write to w. The validation is also done before T_Y writes to w itself (by Lemma 1). Hence, each validation $Rv(T_Y, w)$ by T_Y will observe the value of w that is written by the transaction with the latest commit order before T_Y, which is the same value of w in the memory state right after T_{Y-1} commits ($M^L_{R,Y}$). Since this applies to the validation of every w in $M_{R,Y}$, the validation operations $Rv(T_Y, w)$ for all w in $M_{R,Y}$ are comparing against the same consistent view of global memory. □

Comment: This can be explained in a simpler way via the commit IDs. Each commit unit ensures that all validation operations of a transaction T_X with CID = X are reading from a memory state equivalent to the one right after T_{X-1} commits. Hence, T_X is validating against a consistent view of memory.

8 Logical Indivisibility of Validation and Commit

Kilo TM is designed to permit non-conflicting transactions to commit in parallel. This means that memory state M^L would likely be advanced to another memory state $M^{L'}$ by the commits of other transactions during the validation of T_X and before T_X can commit. This seems to violate Assumption 1. However, with the per-word operation ordering illustrated

by Lemma 1 and 2, we can construct a logical timeline in which validation and commit of each transaction are indivisible.

To construct such a logical timeline, we first define the validation and write operations of a committed transaction T_X as a mini-transaction:

$C_X = Rv(r_1) \dots Rv(r_m)\ W(w_1) \dots W(w_n)$

Each transaction corresponds to a single mini-transaction. The commit order for each mini-transaction is the same as its transaction counterpart. We show that operations performed by these mini-transactions satisfy *conflict serializability* [5].

Lemma 4. *The sequence of operations performed by any arbitrary mini-transactions with Kilo TM satisfies conflict serializability.*

Proof. Let be C_X and C_Y be two mini-transactions with $C_X <_t C_Y$.

C_X has read-set $M_{R,X}$ and write-set $M_{W,X}$, and $M_{A,X} = M_{R,X} \cup M_{W,X}$.

C_Y has read-set $M_{R,Y}$ and write-set $M_{W,Y}$, and $M_{A,Y} = M_{R,Y} \cup M_{W,Y}$.

Each memory location w that is accessed by both transactions (i.e. w is in $(M_{A,X} \cap M_{A,Y})$) will have the operations ordered according to the ordering defined by Lemma 1 and Lemma 2. In all cases, the operations will produce the conflict relation (denoted by ordered pair $(Op(C_A, w), Op(C_B, w))$ below, see definition 3.12 in Weikum and Vossen [5]) that aligns with the commit order. Each conflict relation in turn creates directed edges for their corresponding mini-transactions in a conflict graph (denoted by ordered pair (C_A, C_B) below, see definition 3.15 in Weikum and Vossen [5]):

- $Rv(C_X, w)$ and $W(C_Y, w)$ are always ordered in $Rv(C_X, w) <_P W(C_Y, w)$, producing conflict relation $(Rv(C_X, w), W(C_Y, w))$ and conflict graph directed edge (C_X, C_Y).
- $W(C_X, w)$ and $W(C_Y, w)$ are always ordered in $W(C_X, w) <_P W(C_Y, w)$, producing conflict relation $(W(C_X, w), W(C_Y, w))$ and conflict graph directed edge (C_X, C_Y).
- $W(C_X, w)$ and $Rv(C_Y, w)$ are always ordered in $W(C_X, w) <_P Rv(C_Y, w)$, producing conflict relation $(W(C_X, w), Rv(C_Y, w))$ and conflict graph directed edge (C_X, C_Y).
- $Rv(C_X, w)$ and $Rv(C_Y, w)$ are freely ordered, but they produce no conflict relation.

Since every pair of mini-transactions has either no conflict, or produces conflict relation that aligns with the commit order, the conflict graph created from the conflict relations among all mini-transactions will not contain any cycle. Specifically, two transactions C_A and C_B with $C_A <_t C_B$ can never have a directed path from C_B to C_A in the conflict graph. Therefore, by theorem 3.10 in Weikum and Vossen [5], any sequence of operations performed by the mini-transactions satisfies conflict serializability. □

By Lemma 4 and the definition of conflict serializability (definition 3.14 in Weikum and Vossen [5]), we can imply that any sequence of operations performed by the mini-transactions with Kilo TM has a logically equivalent serial sequence. In this serial sequence, operations performed by each mini-transaction are not interleaved by those from other mini-transactions. This serial sequence forms a logical timeline in which validation and commit of each transaction are indivisible.

9 Tolerance to ABA Problem (Kilo TM)

With Lemma 3 and Lemma 4 proving how Kilo TM satisfies Assumption 2 and Assumption 1 for Theorem 1, Theorem 2 follows:

Theorem 2. *Kilo TM can tolerate the ABA problem.*

Proof. Lemma 3 and Lemma 4 show that Kilo TM satisfies both assumptions for Theorem 1. Hence, Kilo TM can tolerate the ABA problem. □

References

[1] W. W. L. Fung et al., "Hardware Transactional Memory for GPU Architectures," in *Proc. 44th Int'l Symp. on Microarchitecture (MICRO '11)*, pp. 296–307, ACM, 2011.

[2] M. Olszewski et al., "JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory," in *Int'l Conf. on Parallel Architecture and Compilation Techniques*, 2007.

[3] L. Dalessandro et al., "NOrec: Streamlining STM by Abolishing Ownership Records," in *Proc. 15th Symp. on Principles and Practices of Parallel Programming (PPoPP 10)*, pp. 67–78, ACM, 2010.

[4] M. M. Michael, "Practical Lock-Free and Wait-Free LL/SC/VL Implementations Using 64-Bit CAS," in *Proc. 18th Int'l Symp. on Distributed Computing (DISC 04)*, pp. 144–158, Springer, 2004.

[5] G. WeiKum and G. Vossen, *Transactional Information Systems: Theory Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

[6] S. Doherty et al., "DCAS is not a Silver Bullet for Nonblocking Algorithm Design," in *Symp. on Parallel Algorithms and Architectures*, 2004.

[7] C. Blundell, E. C. Lewis, and M. M. K. Martin, "Deconstructing Transactional Semantics: The Subtleties of Atomicity," in *WDDD*, 2005.

[8] M. F. Spear et al., "RingSTM: Scalable Transactions with a Single Atomic Instruction," in *Proc. 20thSymp. on Parallel Algorithms and Architectures (SPAA 08)*, pp. 275–284, ACM, 2008.